CS 444 Operating Systems

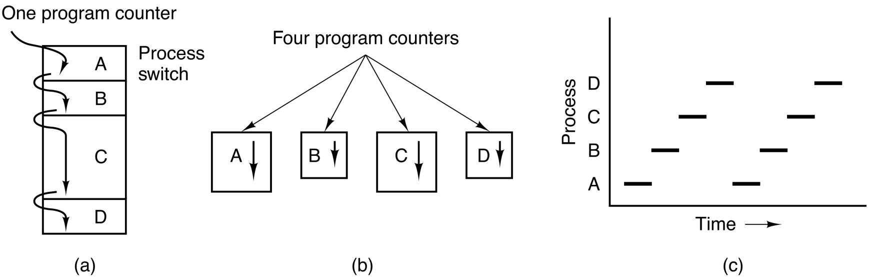
Chapter 2 Processes and Threads

J. Holly DeBlois

September 12, 2024

The Process Model

- A (sequential) process is an instance of running a program
- Multiprogramming of four processes is done through process switch
- These processes are independent in appearance
- Only one process is active at a time



Process Creation

- Four principal events that cause processes to be created

1. System initialization (ssh, printer, mail, web daemons)
2. Execution of a process creation system call by a running process
    - `fork()`
3. A user request to create a new process
4. Initiation of a batch job

POSIX System Calls for Process Creation

- `fork()`: duplicate the core image
- `execve()`: wipe the core image and load another executable
  - The entry in the process table is unchanged
  - Maintain the process tree, which is rooted at process 1
- Reason for this two-step process creation:

  Allow the child process to manipulate the file descriptors after `fork()` so that `stdin`, `stdout`, and `stderr` can be redirected

  For example,
  - Redirect `stdin` to a data file
  - Redirect `stdout` to a printer
  - Redirect `stderr` to a log file
  - Redirect `stdin` and `stdout` to a pipe

Process Termination

- Typical conditions that terminate a process:

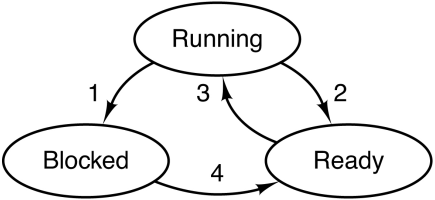
1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

Process States

- Three process states

1. Running, actually using the CPU at that instant
2. Ready, runnable, temporarily stopped to let another process run
    - Ready queue
    - Process scheduler
3. Blocked, unable to run until some external event happens

Process States



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

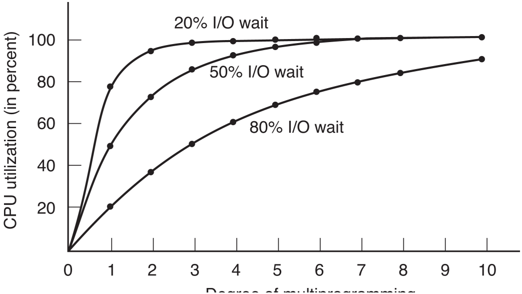
Process Table

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment info Pointer to data segment info Pointer to stack segment info	Root directory Working directory File descriptors User ID Group ID

Handling Interrupts

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly-language procedure saves registers.
4. Assembly-language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly-language procedure starts up new current process.

Modeling Multiprogramming

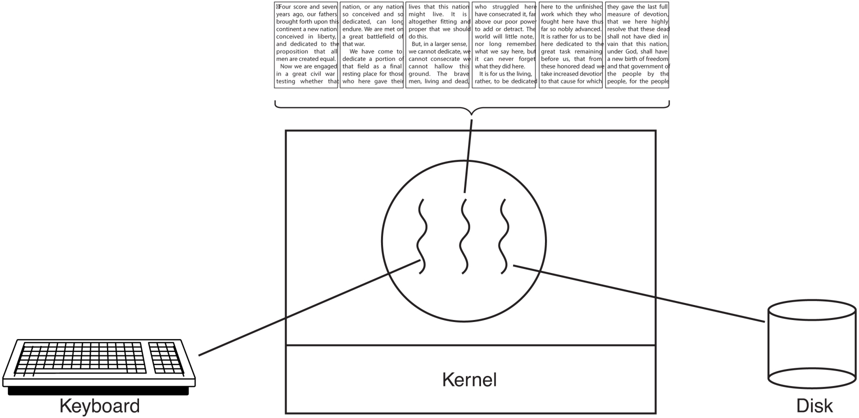


- System performance depends on degree of multiprogramming
- CPU utilization $= 1 - p^n$
- Queueing theory

Example of Degree of Multiprogramming

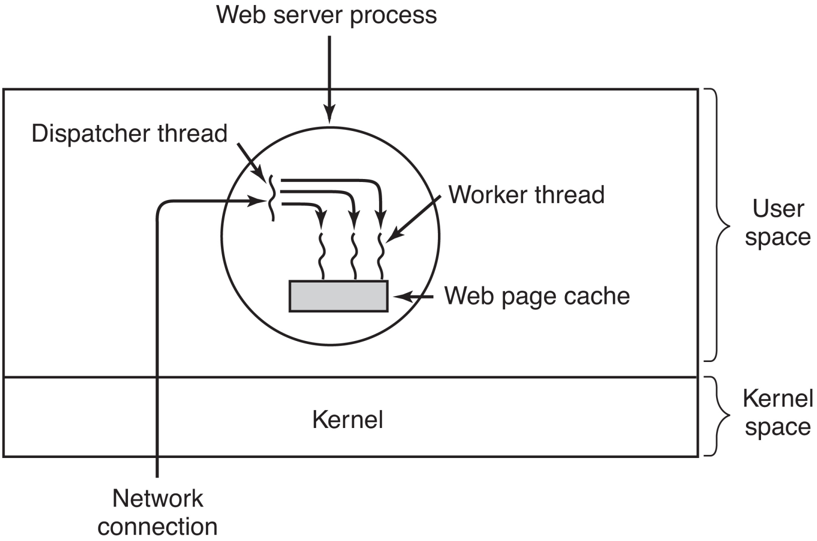
- 8GB RAM: OS takes 2GB; run 3 processes at 2GB each
  - 80% I/O wait leads to 49% CPU utilization
- Add 8GB RAM, run 4 additional processes
  - 79% CPU utilization
- Add yet another 8GB RAM, run 4 additional processes
  - 91% CPU utilization

Thread Usage: Word Processor



Thread Usage: Web Server

- Thread creation is 10 to 100 times faster than process creation



Web Server: Dispatcher and Worker Threads

- Master/slave programming model
- (a) Dispatcher thread will block if no request
- (b) Worker thread will block if no work

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

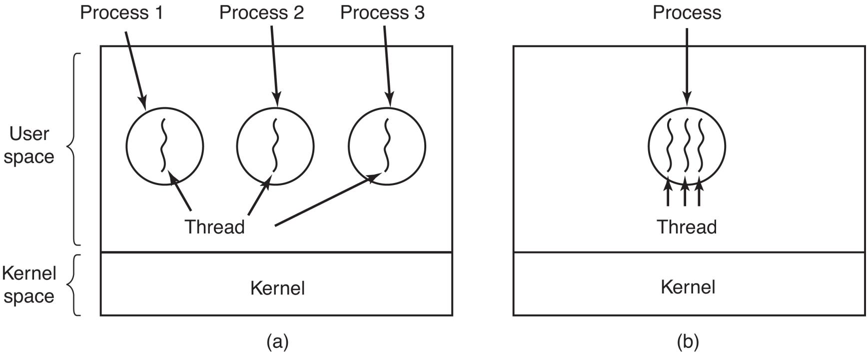
Three Programming Models of Servers

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

- Threads: easy to code, good performance
- Single-threaded process: easy to code, low performance
- Finite-state machine: hard to code, high performance

The Classical Thread Model

- 3 single-threaded processes
- 1 process with 3 threads

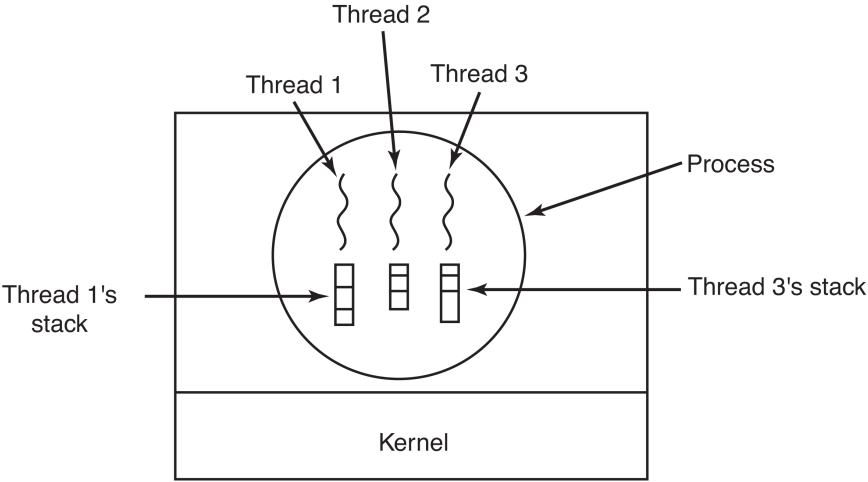


Process and Thread Data

- Processes are used to group resources together
- Threads are the entities scheduled for execution on the CPU

Per-process items	**Per-thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Each Thread Has its Own Stack



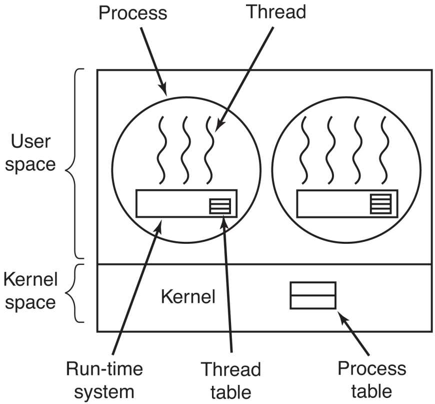
POSIX Threads

- Tutorial: `https://hpc-tutorials.llnl.gov/posix/`
- We will use Pthreads in a programming project later

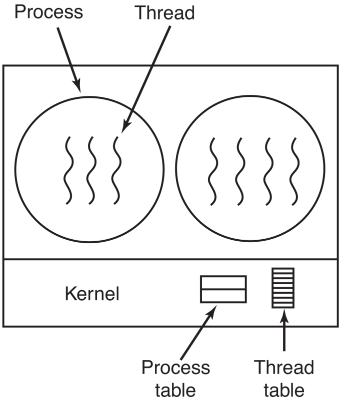
Thread call	**Description**
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Implementing Threads at the User- or Kernel-Level

- A user-level thread library



- A kernel-level thread package



User-Level Thread Library

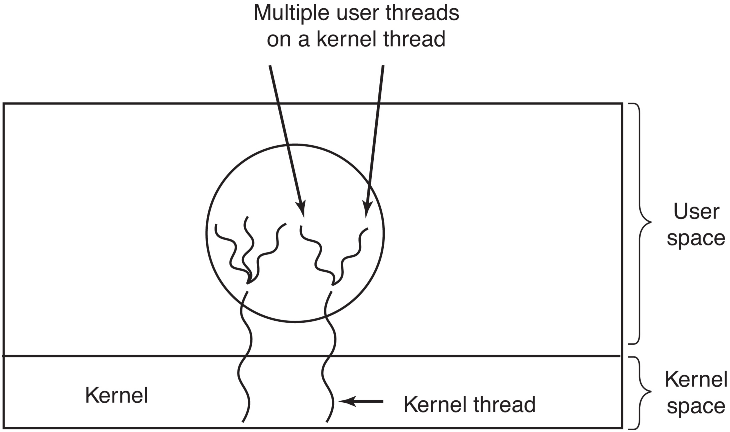
- Pros
  - Faster thread switching
  - Customized thread scheduling
- Cons
  - How to handle a blocking system call made by one thread
  - What to do when one thread incurs page fault
  - The original reason to do multithreading is for applications where threads block often

Issues in Kernel-Level Threads

- After `fork()`, should the threads be duplicated?
- Which threads handle signals?

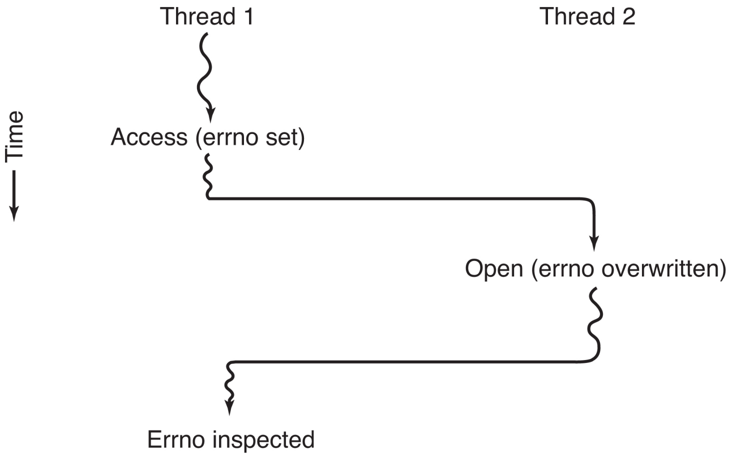
Hybrid Implementation

- Multiplexing user-level threads onto kernel-level threads
- In Linux, one user thread is mapped to one kernel thread



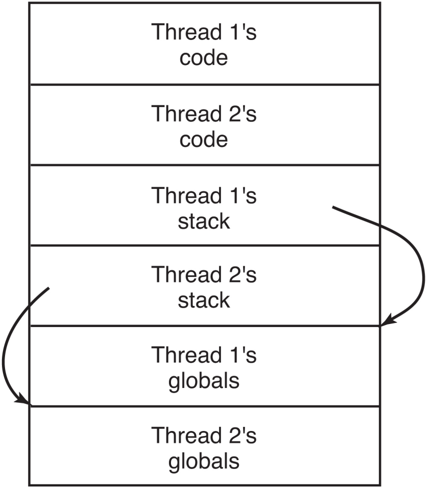
Issues in Making Single-Threaded Code Multithreaded

- Conflicts between threads over the use of a global variable



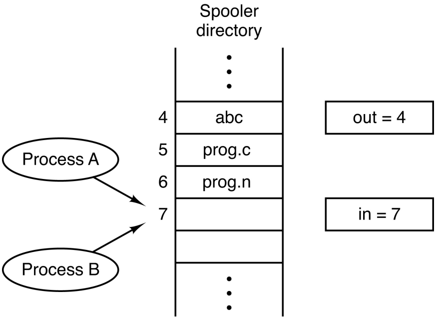
Private Global Variables

- Let threads have private global variables



Race Conditions

- Two processes want to access shared memory at the same time
- The results depend on who wins and who loses the race

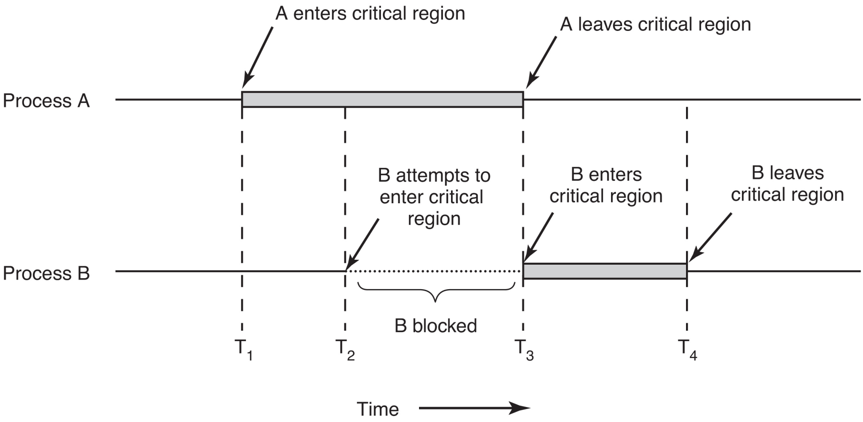


Requirements to Avoid Race Conditions

- No two processes may be simultaneously inside their critical regions
- No assumptions may be made about speeds or the number of CPUs
- No process running outside its critical region may block other processes
- No process should have to wait forever to enter its critical region

Critical Regions

- Mutual exclusion when using critical regions



Mutual Exclusion by Busy Waiting

- Busy waiting at the inner while loops — note the semicolons at end of line
- This implements strict alternation
- Not ideal

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Peterson's Solution, Software Busy Waiting

- First software solution for mutual exclusion without strict alternation

```
#define FALSE  0
#define TRUE   1
#define N      2                            /* number of processes */

int turn;                                   /* whose turn is it? */
int interested[N];                          /* all values initially 0 (FALSE) */

void enter_region(int process);             /* process is 0 or 1 */
{
      int other;                            /* number of the other process */

      other = 1 – process;                  /* the opposite of process */
      interested[process] = TRUE;           /* show that you are interested */
      turn = process;                       /* set flag */
      while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)              /* process: who is leaving */
{
      interested[process] = FALSE;          /* indicate departure from critical region */
}
```

Peterson's Solution, 1981

```
#define FALSE 0
#define TRUE 1
#define N 2
int jail;
int interested[N] = {FALSE, FALSE};
void enterRegion(int me) {
  int you;
  you = 1 - me;
  interested[me] = TRUE;
  jail = me;
  while (jail == me && interested[you] == TRUE)
    ;
}
void leaveRegion(int me) {
  interested[me] = FALSE;
}
```

Generalized Peterson's Solution

```
#define N 5
int height[N] = { -1 };
int jail[N];
void leaveRegion(int me) {
  height[me] = -1;
}
```

```
void enterRegion(int me) {
  int level, other;
  for (level = 0; level < N; level++) {
    height[me] = level;
    jail[level] = me;
    while (jail[level] == me) {
      for (other = 0; other < N; other++) {
        if (other == me)
          continue;
        if (height[other] >= level)
          break;
      }
      if (other == N)
        break;
    }
  }
}
```

The TSL Instruction, Hardware Busy Waiting

- Busy waiting with TSL
- Hardware locks the memory bus for exclusive use by the thread
- TSL is an atomic operation, run without interruption

```
enter_region:
      TSL REGISTER,LOCK        | copy lock to register and set lock to 1
      CMP REGISTER,#0          | was lock zero?
      JNE enter_region         | if it was not zero, lock was set, so loop
      RET                      | return to caller; critical region entered

leave_region:
      MOVE LOCK,#0             | store a 0 in lock
      RET                      | return to caller
```

The Intel XCHG Instruction

- Reduces the duration that XCHG locks the memory bus

```
enter_region:
    MOVE REGISTER,#1          | put a 1 in the register
    XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
    CMP REGISTER,#0           | was lock zero?
    JNE enter_region          | if it was non zero, lock was set, so loop
    RET                       | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0              | store a 0 in lock
    RET                       | return to caller
```

Problems with Busy Waiting

- If a low priority thread is holding the lock, and a high priority thread is busy waiting
- Priority inversion
- The low priority thread doesn't get the CPU and can't release the lock
- They may not get out of this scenario
- Solution: the high priority thread should go to sleep instead of busy waiting

A Non-Solution for the Producer-Consumer Problem

- A fatal race condition with lost wakeup calls

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Semaphore

- Nonnegative integers store the number of wakeup calls
- `up()` and `down()` are atomic operations
- All participants must follow a prescribed sequence of `down()` calls
- The order of `down()` calls is important

A Semaphore Solution for the Producer-Consumer Problem

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
void producer(void) {
  int item;
  while (1) {
    item = produce();
    down(&empty);
    down(&mutex);
    insert(item);
    up(&mutex);
    up(&full);
  }
}
```

```
void consumer(void) {
  int item;
  while(1) {
    down(&full);
    down(&mutex);
    item = delete();
    up(&mutex);
    up(&empty);
    consume(item);
  }
}
```

Implementations of Mutex by TSL without Busy Waiting

- `mutex_lock` and `mutex_unlock`
- Linux uses futex, fast user space mutex

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Monitors

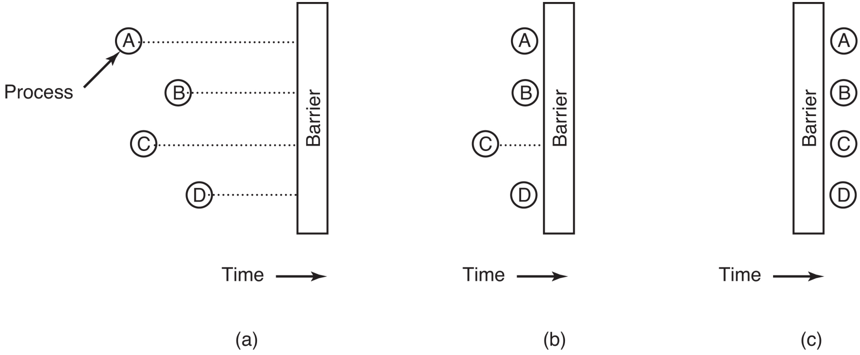
```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .        .        .
        end;
end monitor;
```

- A programming language construct
- Absent in C
- Available in Java
- Java provides user-level threads
- Java keyword synchronized

Barrier: A Library Procedure

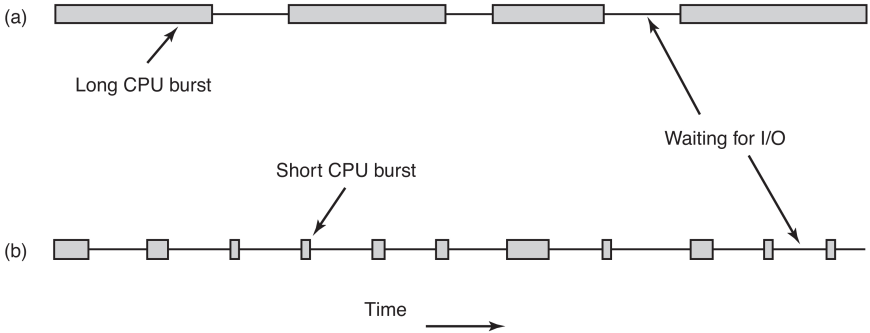


- Processes (threads) approach a barrier
- When the last process arrives at the barrier, all of them are let through
- Barriers are typically used at the end of a loop to synchronize the threads that run the loop in parallel

Importance of Process Scheduling

- Mainframes, running both batch and interactive processes: very important
- PC: not important at all
- Networked servers: important
- Mobile devices, sensor nodes?
- Future: scheduling to reduce power consumption?

Characteristics of Processes



- CPU-bound
- I/O-bound
- Characterized by lengths of CPU burst

Categories of Scheduling Algorithms

- Batch
- Interactive
- Real time
- Preemptive
- Nonpreemptive

Goals of Scheduling Algorithms

All systems

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems

Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Performance Metrics

- Throughput: number of jobs finished per unit time
- Turnaround time: in a batch system, the average time from submission to completion
- CPU utilization
- Response time: in an interactive system, the time between issuing a command and getting the result
  - Proportionality: user expectation
- Waiting time: elapsed time minus CPU time

Scheduling in Batch Systems

- First-come first-served
- Shortest job first
  - Nonpreemptive, optimal
- Shortest remaining time next
  - Preemptive

Shortest Job First

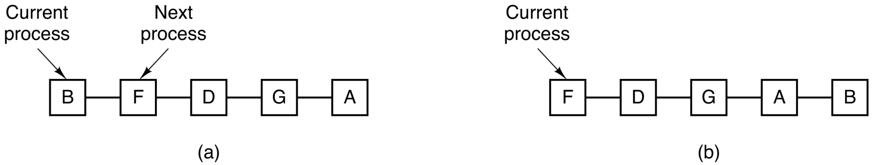
- Running four jobs in FCFS
- Running them in shortest job first order

Scheduling in Interactive Systems

- Round-robin scheduling
- Priority scheduling
  - Unix command `nice`
  - Multiple priority queues, use FCFS within same priority
- Shortest process next
  - Estimated, aging of past measurements
- Guaranteed scheduling
  - Equal amount of CPU time
- Lottery scheduling
- Fair-share scheduling
  - Users, not processes, get equal amount of CPU time

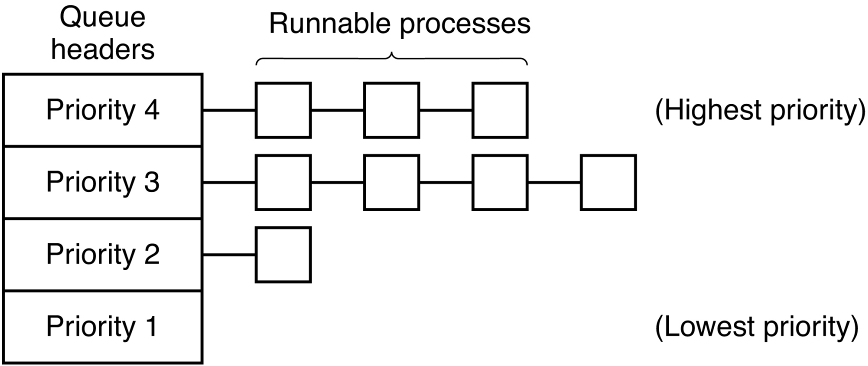
Round-Robin Scheduling

- Run the first process in the queue for a fixed quantum
- Append it to the end of queue after it uses up its quantum



Priority Scheduling

- A scheduling algorithm with four priority levels
- Linux: 140 priority levels, 0 highest, 139 lowest



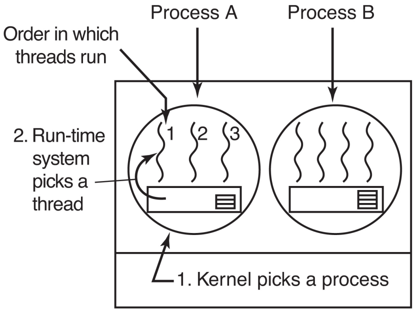
Scheduling in Real-Time Systems

- Time plays an essential role
- Categories
  - Hard real time
  - Soft real time
  - Periodic or aperiodic
- m periodic processes, process i occurs with period P_i and takes C_i sec of CPU time
- These m processes are schedulable if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

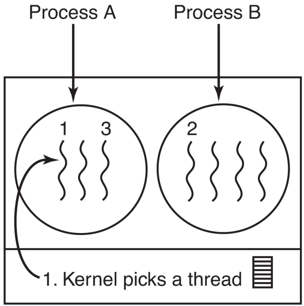
Thread Scheduling

- User-level threads



Possible: A1, A2, A3, A1, A2, A3
Not possible: A1, B1, A2, B2, A3, B3
(a)

- Kernel-level threads

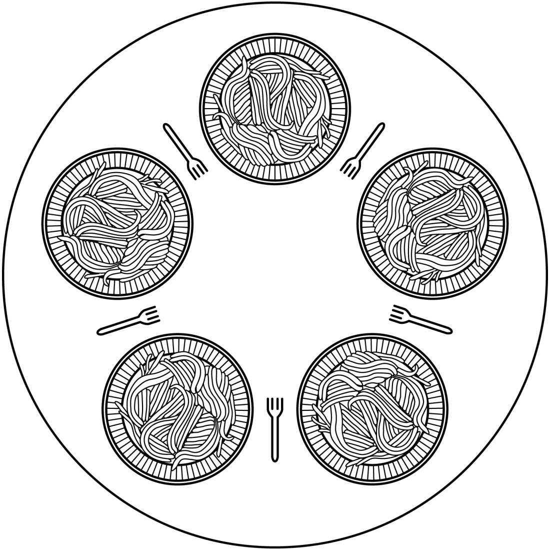


Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3
(b)

Separation of Scheduling Policy and Mechanism

- Key idea not widely implemented
- Scheduling mechanism must remain in the OS kernel
- Allow scheduling policy to be parameterized by user processes
- Example: database management system

The Dining Philosophers Problem



A Non-solution to the Dining Philosophers Problem

```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                    /* philosopher is thinking */
        take_fork(i);                /* take left fork */
        take_fork((i+1) % N);        /* take right fork; % is modulo operator */
        eat( );                      /* yum-yum, spaghetti */
        put_fork(i);                 /* put left fork back on the table */
        put_fork((i+1) % N);         /* put right fork back on the table */
    }
}
```