Processes and Threads

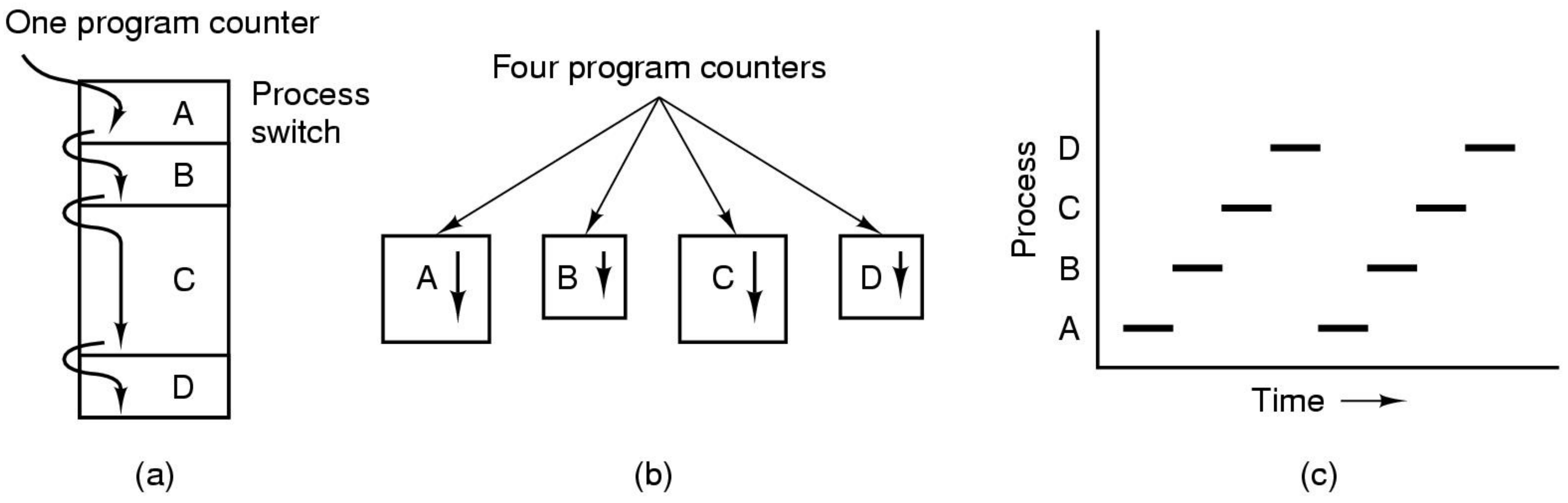
Instructor: Dr. Liting Hu

What is a process?

- An instance of a program, replete with registers, variables, and a program counter
- It has a program, input, output and a state
- Why is this idea necessary?

A computer manages many computations concurrently - need an abstraction to describe how it does it

Multiprogramming



(a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

Process Creation

Events which can cause process creation

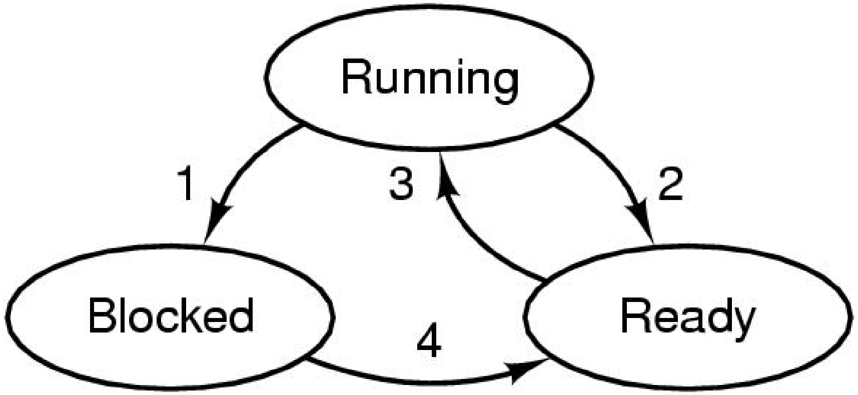
- System initialization.
- Execution of a process creation system call by a running process.
- A user request to create a new process.
- Initiation of a batch job.

Process Termination

Events which cause process termination:

- Normal exit (voluntary).
- Error exit (voluntary).
- Fatal error (involuntary).
- Killed by another process (involuntary).

Process States

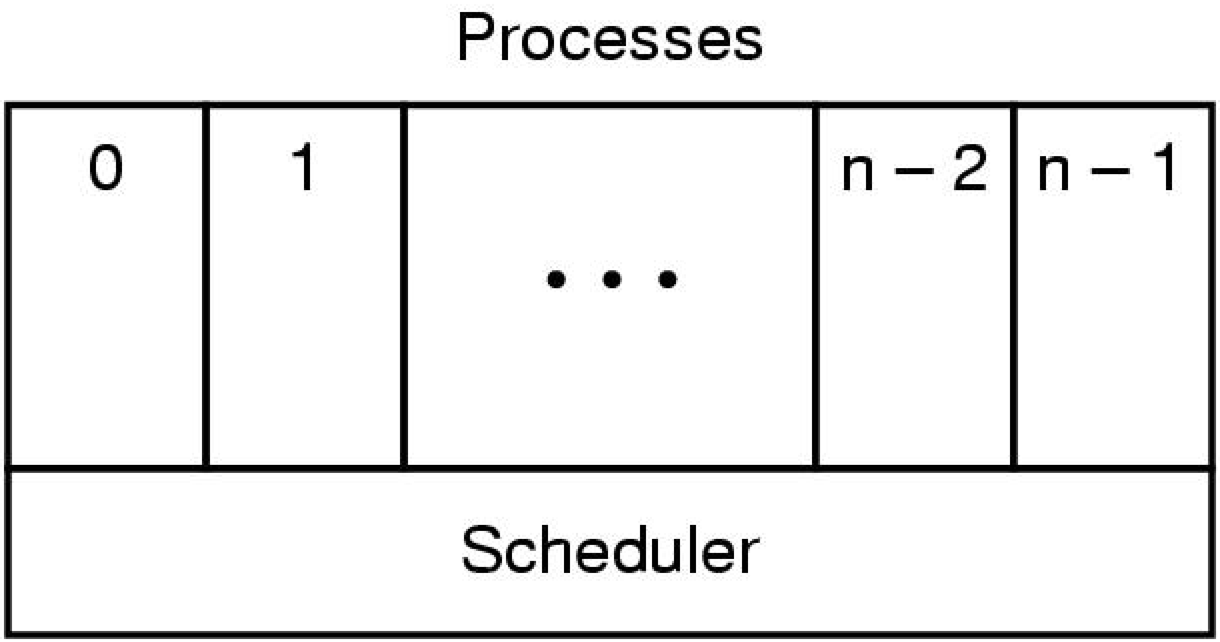


1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

cat chapter1 chapter2 chapter3 | grep tree

A process can be in running, blocked, or ready state. Transitions between these states are as shown.

Implementation of Processes (1)



The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

Implementation of Processes (2)

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
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		

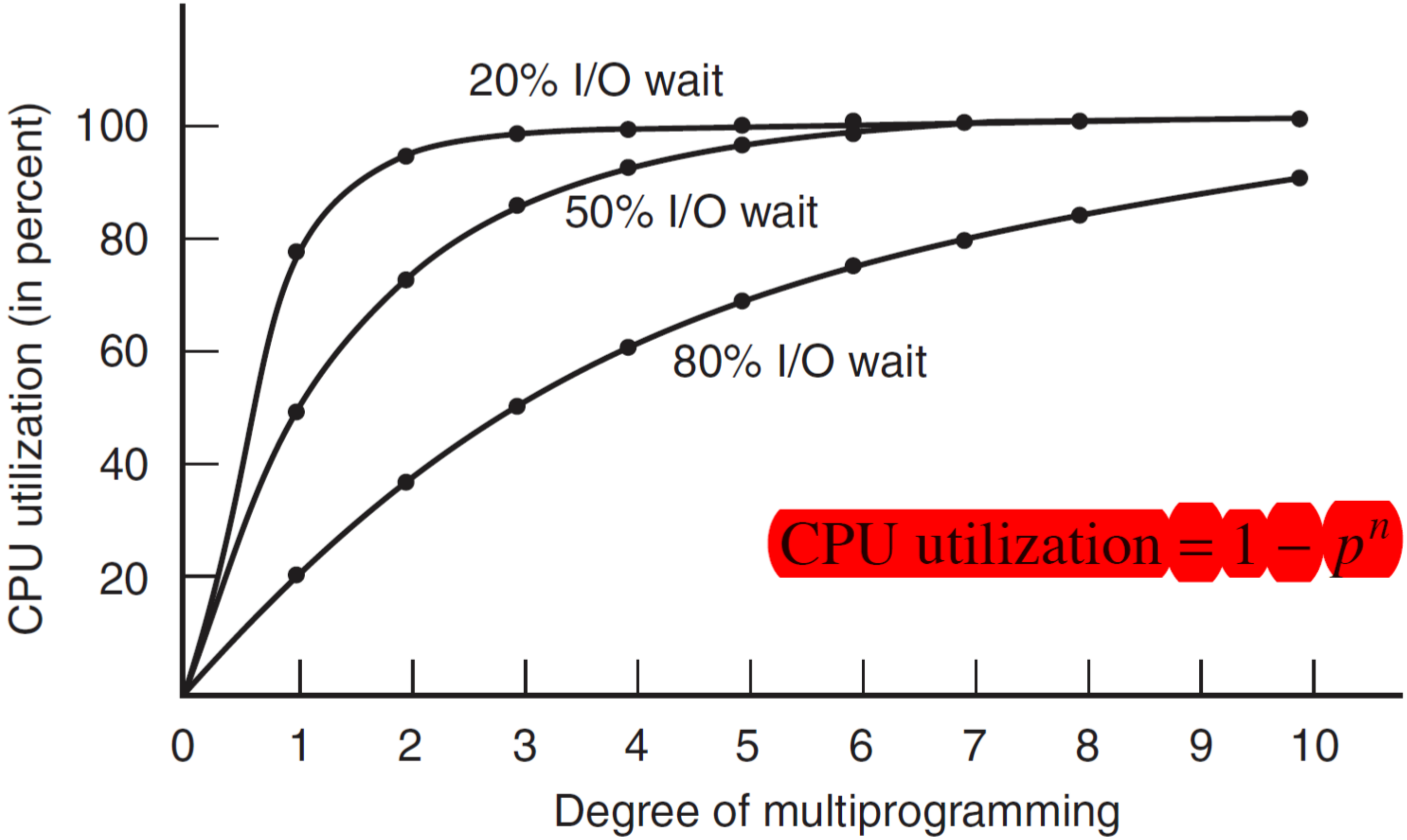
Some of the fields of a typical **process table entry.**

OS processes an interrupt

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.

Skeleton of what **the lowest level of the operating system** does when an interrupt occurs.

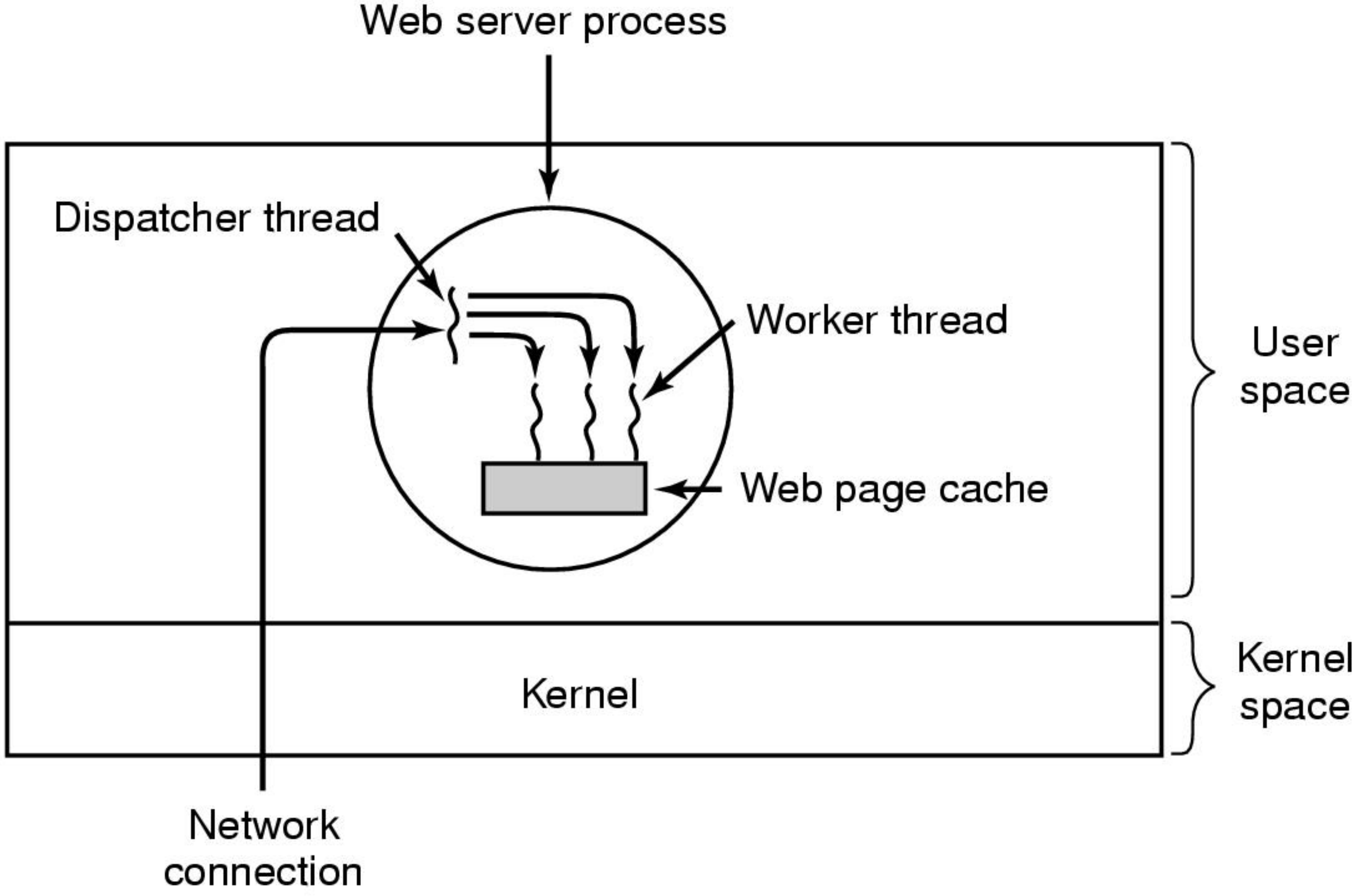
How multiprogramming performs



$$\text{CPU utilization} = 1 - p^n$$

CPU utilization as a function of the number of processes in memory.

Thread Example-web server



Web server

- If page is not there, then thread blocks
- CPU does nothing while it waits for page
- Thread structure enables server to instantiate another page and get something done

Threads are lightweight

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	

Threads are like processes

- Have same states
  - Running
  - Ready
  - Blocked
- Have their own stacks –same as processes
  - Stacks contain frames for (un-returned) procedure calls
    - Local variables
    - Return address to use when procedure comes back

How do threads work?

- Start with one thread in a process
- Thread contains (id, registers, attributes)
- Use library call to create new threads and to use threads
  - Thread_create includes parameter indicating what procedure to run
  - Thread_exit causes thread to exit and disappear (can't schedule it)
  - Thread_join Thread blocks until another thread finishes its work
  - Thread_yield Release the CPU to let another thread run

POSIX Threads (Pthreads)

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

Pthreads are IEEE Unix standard library calls

A Pthreads example-”Hello,world”

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
        /* This function prints the thread's identifier and then exits. */
        printf("Hello World. Greetings from thread %d0, tid);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        /* The main program creates 10 threads and then exits. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Main here. Creating thread %d0, i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

                if (status != 0) {
                        printf("Oops. pthread_create returned error code %d0, status);
                        exit(-1);
                }
        }
        exit(NULL);
}
```

Interprocess Communication

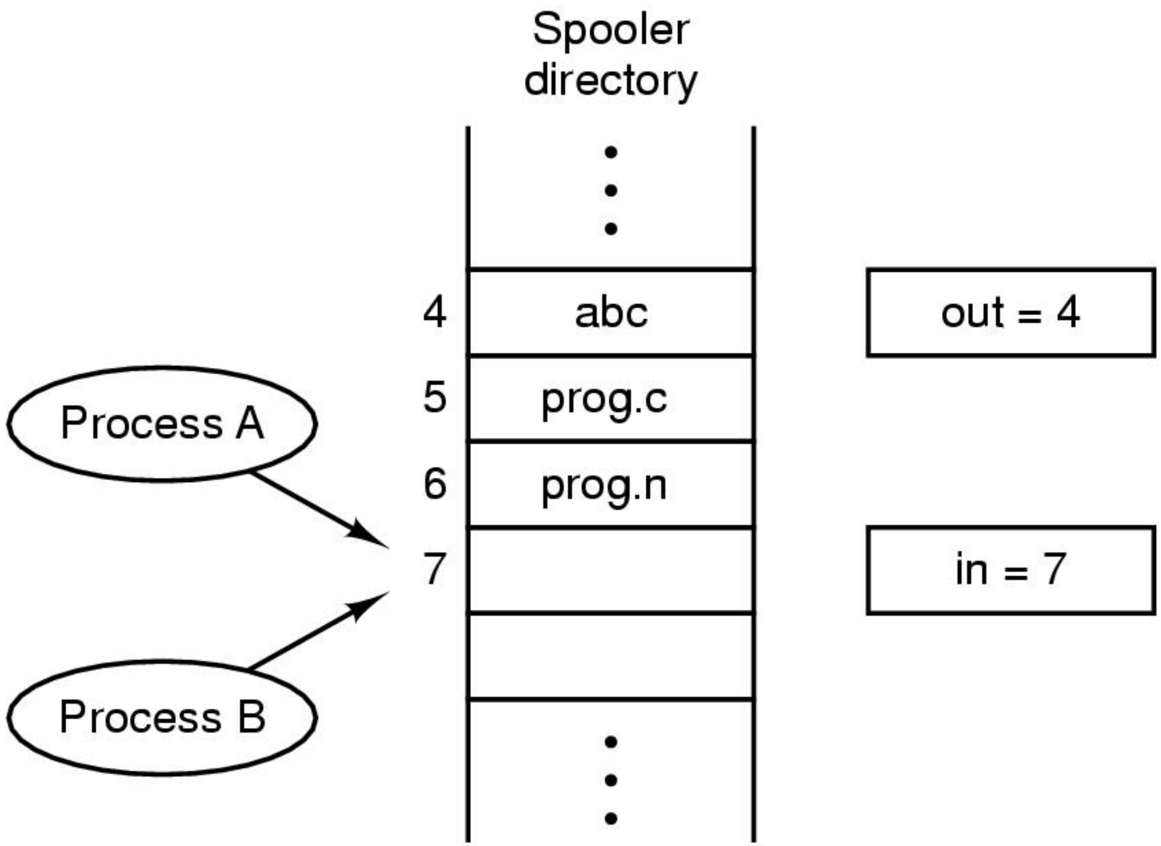
- Three problems
  - How to actually do it
  - How to deal with process conflicts (2 airline reservations for same seat)
  - How to do correct sequencing when dependencies are present-aim the gun before firing it

- SAME ISSUES FOR THREADS AS FOR PROCESSES-SAME SOLUTIONS AS WELL

Race Conditions

- Real-life example
- https://www.youtube.com/watch?v=8zyYjlaEY1k

Race Conditions



In is local variable containing pointer to next free slot

Out is local variable pointing to next file to be printed

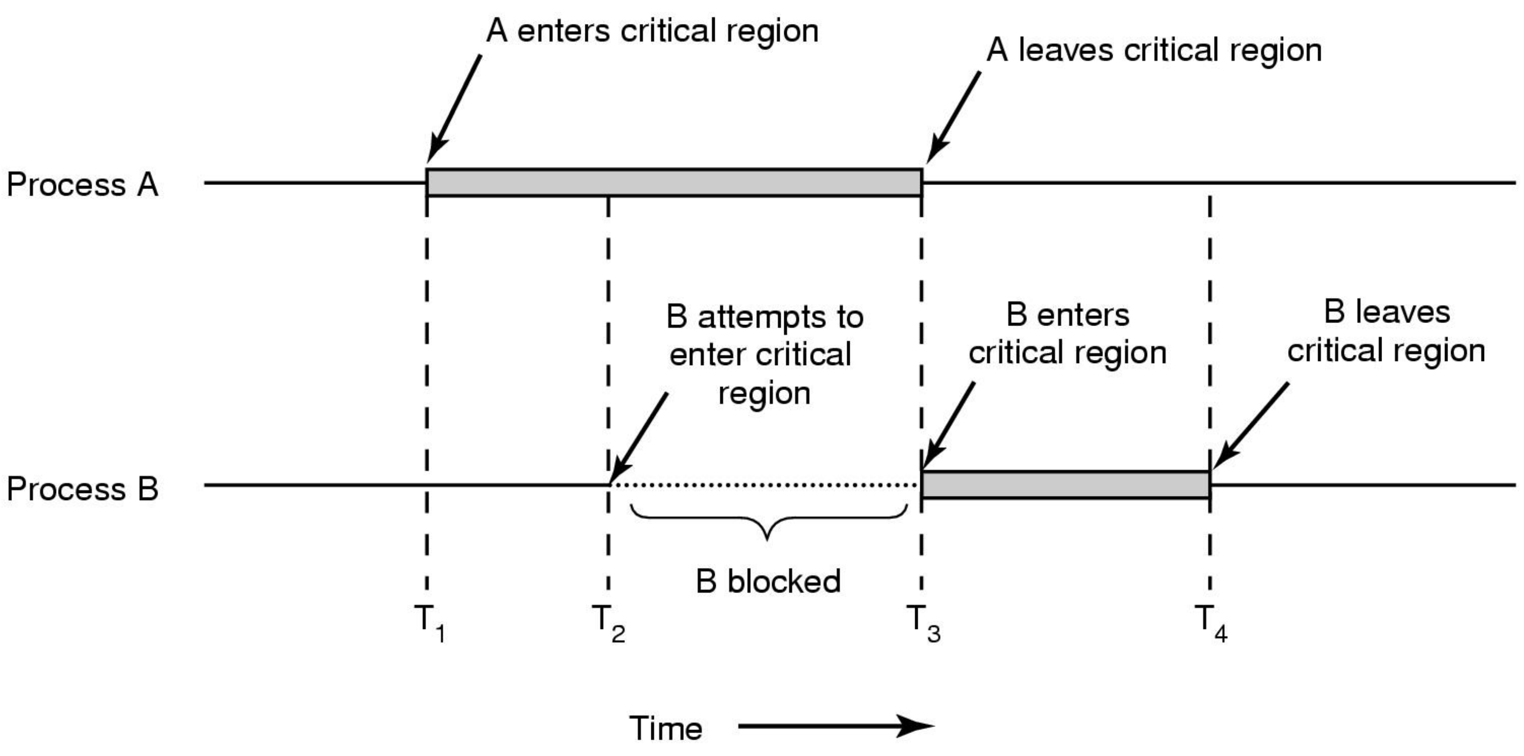
How to avoid races

- Mutual exclusion–only one process at a time can use a shared variable/file
- Critical regions-shared memory which leads to races
- Solution- Ensure that two processes can't be in the critical region at the same time

Properties of a good solution

- Mutual exclusion
- No assumptions about speeds or number of CPU's
- No process outside critical region can block other processes
- No starvation - no process waits forever to enter critical region

What we are trying to do



First attempts-Busy Waiting

A laundry list of proposals to achieve mutual exclusion

- Disabling interrupts
- Lock variables
- Strict alternation
- Peterson's solution
- The TSL instruction

Disabling Interrupts

- Idea: process disables interrupts, enters critical region, enables interrupts when it leaves critical region

- Problems
  - Process might never enable interrupts, crashing system
  - Won’t work on multi-core chips as disabling interrupts only effects one CPU at a time

Lock variables

- A software solution-everyone shares a lock
  - When lock is 0, process turns it to 1 and enters critical region
  - When exit critical region, turn lock to 0

- Problem-Race condition

Strict Alternation

```
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)

First me, then you

Problems with strict alternation

- Employs busy waiting-while waiting for the critical region (cr), a process spins

- If one process is outside the cr and it is its turn, then other process has to wait until outside guy finishes both outside AND inside (cr) work

Peterson's Solution

```
#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

- Process 0 & 1 try to get in simultaneously
- Last one in sets turn: say it is process 1
- Process 0 enters (turn= = process is False)

TSL

- TSL reads lock into register and stores NON ZERO VALUE in lock (e.g. process number)
- Instruction is atomic: done by freezing access to bus line (bus disable)

What’s wrong with Peterson, TSL?

- Busy waiting - waste of CPU time!

- Idea: Replace busy waiting by blocking calls
  - Sleep blocks process
  - Wakeup unblocks process

The Producer-Consumer Problem (aka Bounded Buffer Problem)

- https://www.youtube.com/watch?v=GvfjiA9jkTs

The Producer-Consumer Problem (aka Bounded Buffer Problem)

```
#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 */
     }
}
```

Semaphores

- Semaphore is an integer variable
- Used to sleeping processes/wakeups
- Two operations, down and up
- Down checks semaphore. If not zero, decrements semaphore. If zero, process goes to sleep
- Up increments semaphore. If more then one process asleep, one is chosen randomly and enters critical region (first does a down)
- **ATOMIC IMPLEMENTATION-**interrupts disabled

Producer Consumer with **Semaphores**

- 3 semaphores: full, empty and **mutex**
- Full counts full slots (initially 0)
- Empty counts empty slots (initially N)
- Mutex protects variable which contains the items produced and consumed

Mutexes

- Don't always need counting operation of semaphore, just mutual exclusion part
- Mutex: variable which can be in one of two states-locked (0), unlocked(1 or other value)
  - Easy to implement
- Good for using with thread packages in user space
  - Thread (process) wants access to cr, calls mutex_lock.
  - If mutex is unlocked, call succeeds. Otherwise, thread blocks until thread in the cr does a mutex_unlock.

Semaphores

- https://www.youtube.com/watch?v=DvF3AsTglUU

Producer Consumer with semaphores

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

Message Passing

- Information exchange between machines
- Two primitives
  - Send(destination, &message)
  - Receive(source,&message)
- Lots of design issues
  - Message loss
  - acknowledgements, time outs deal with loss
  - Authentication-how does a process know the identity of the sender?

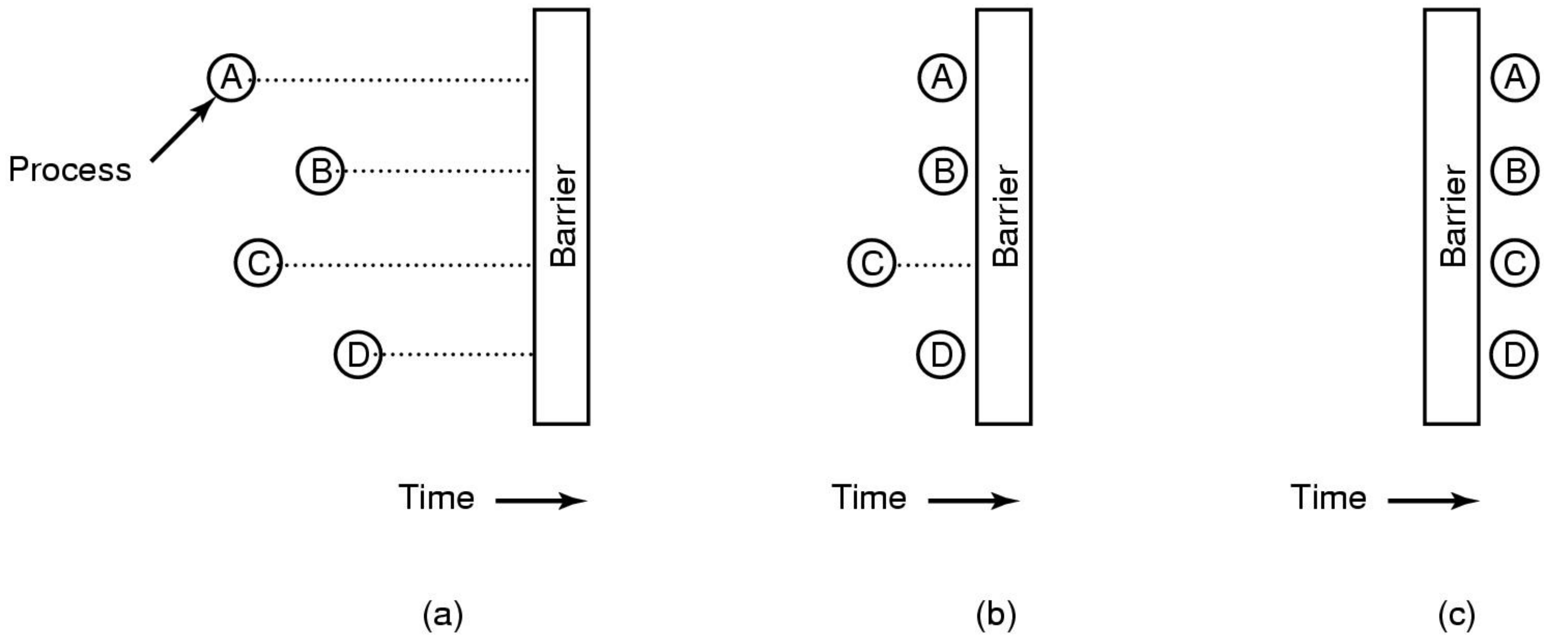
Producer Consumer Using Message Passing

- Consumer sends N empty messages to producer
- Producer fills message with data and sends to consumer

Message Passing Approaches

- Have unique ID for address of recipient process
- Mailbox
  - In producer consumer, have one for the producer and one for the consumer
- No buffering-sending process blocks until the receive happens. Receiver blocks until send occurs (Rendezvous)
- MPI

Barriers



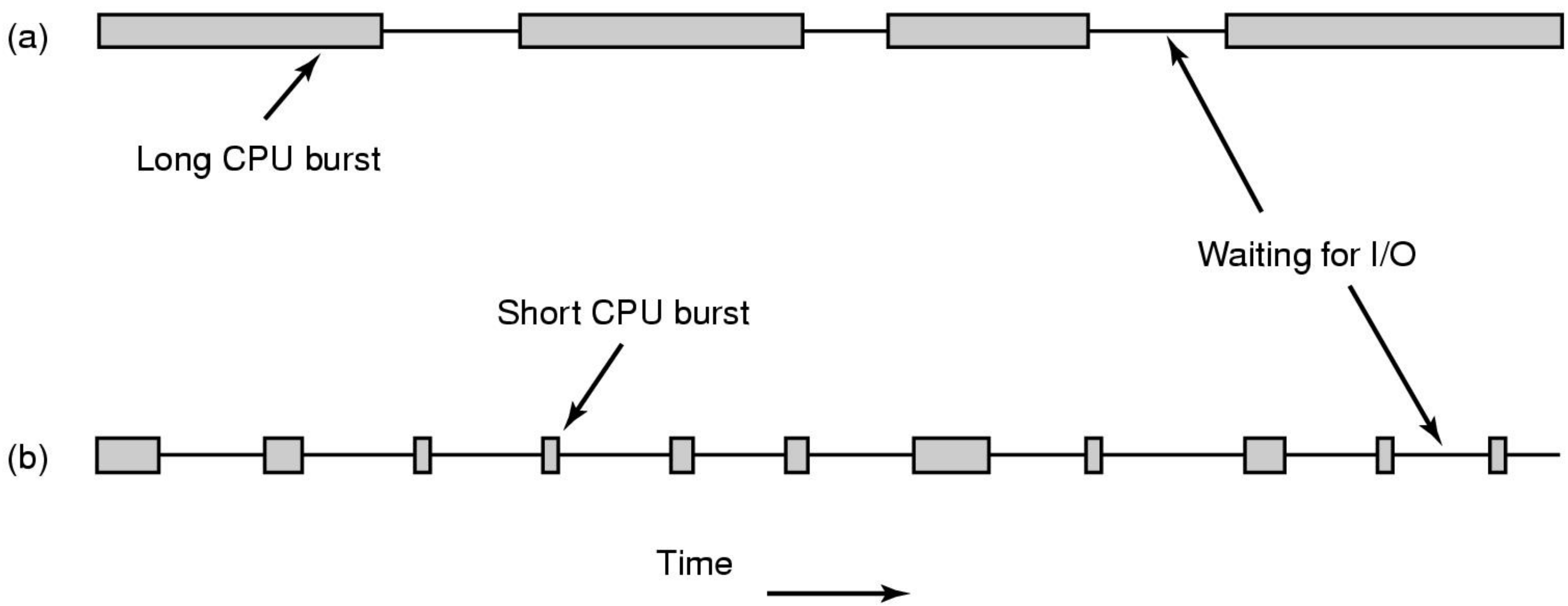
Who cares about scheduling algorithms?

- Batch servers
- Time sharing machines
- Networked servers
- You care if you have a bunch of users and/or if the demands of the jobs differ

Who doesn’t care about scheduling algorithms?

- PC’s
  - One user who only competes with himself for the CPU

Scheduling – Process Behavior



Bursts of CPU usage alternate with periods of waiting for I/O.
(a) A CPU-bound process. (b) An I/O-bound process.

When to make scheduling decisions

- New process creation (run parent or child)
- Schedule when
  - A process exits
  - A process blocks (e.g. on a semaphore)
  - I/O interrupt happens

Categories of Scheduling Algorithms

- Batch (accounts receivable, payroll…..)
- Interactive
- Real time (deadlines)

Depends on the use to which the CPU is being put

Scheduling Algorithm Goals

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

Scheduling in Batch Systems

- First-come first-served
- Shortest job first
- Shortest remaining time next

First come first serve

- Easy to implement
- Won’t work for a varied workload
  - I/O process (long execution time) runs in front of interactive process (short execution time)

Shortest Job First

- Need to know run times in advance
- Non pre-emptive algorithm
- Provably optimal

Eg 4 jobs with runs times of **a,b,c,d**

First finishes at a, second at **a+b**,third at **a+b+c**, last at **a+b+c+d**

Mean turnaround time is **(4a+3b+2c+d)/4**

=> smallest time has to come first to minimize the mean turnaround time

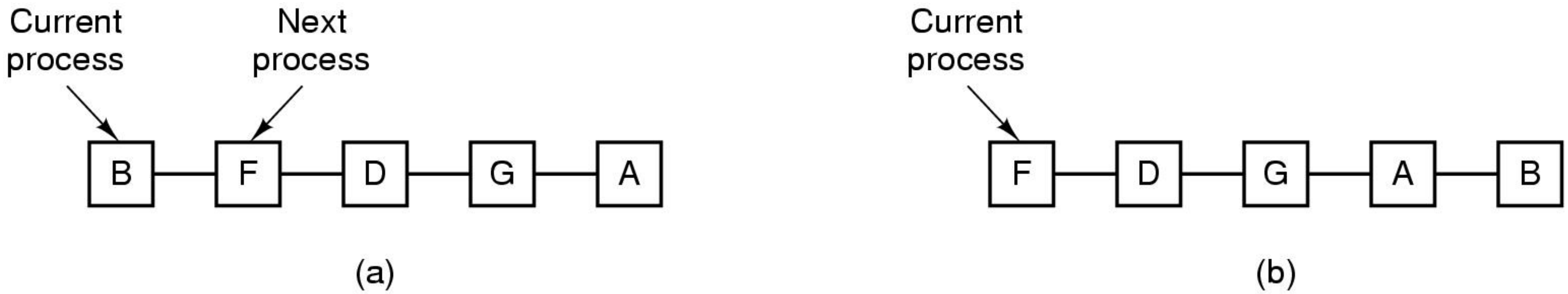
Shortest Remaining Time Next

- Pick job with shortest remaining time to execute next
- Pre-emptive: compare running time of new job to remaining time of existing job
  - Need to know the run times of jobs in advance

Scheduling in Interactive Systems

- Round robin
- Priority
- Multiple Queues
- Shortest Process Next
- Guaranteed Scheduling
- Lottery Scheduling
- Fair Share Scheduling

Round-Robin Scheduling



Process list-before and after

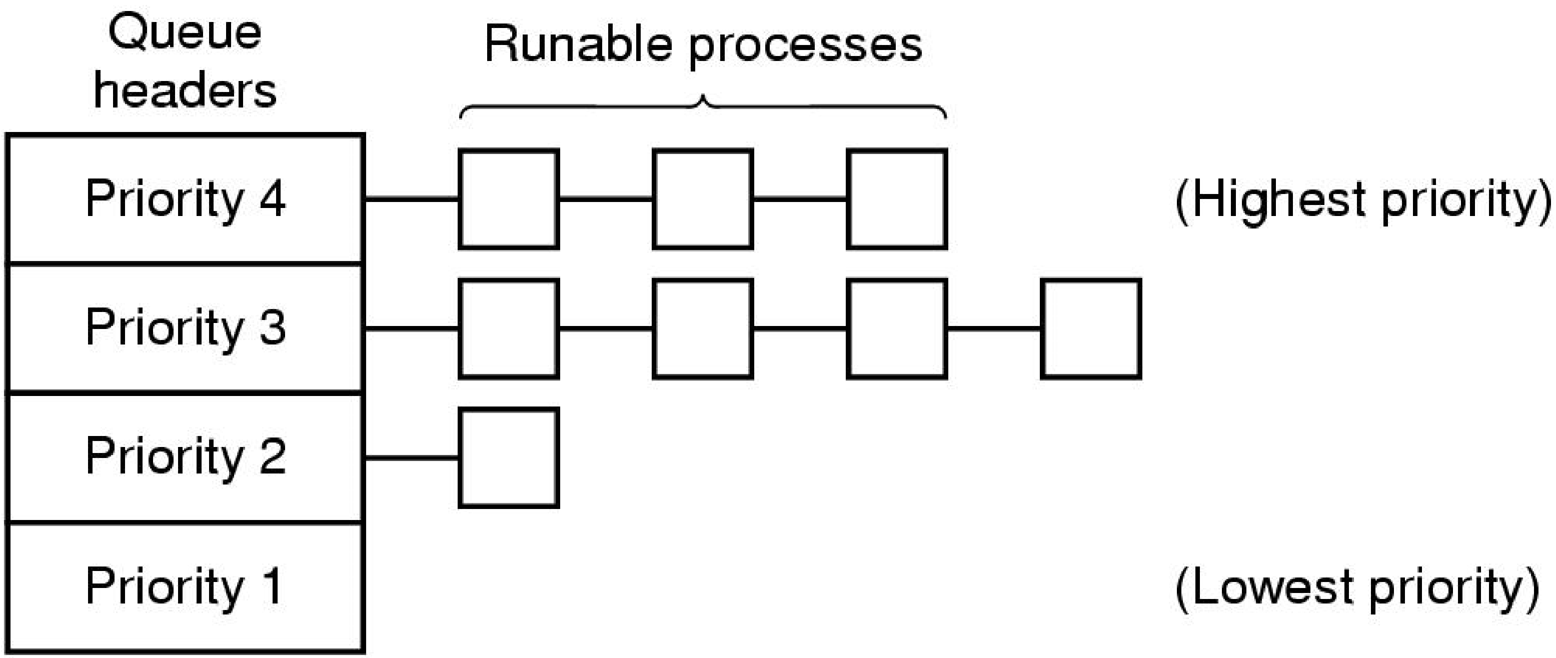
Round robin

- Quantum too short => too many process switches
- Quantum too long => wasted cpu time
  - 20-50 msec seems to be OK
- Don’t need to know run times in advance

Priority Scheduling (1)

- Run jobs according to their priority
- Can be static or can do it dynamically
- Typically combine RR with priority. Each priority class uses RR inside

Priority Scheduling (2)



Multiple Queues with Priority Scheduling

- Highest priority gets one quantum, second highest gets 2…..
  - If highest finishes during quantum, great. Otherwise bump it to second highest priority and so on into the night

- Consequently, shortest (high priority) jobs get out of town first
- They announce themselves-no previous knowledge assumed!

Shortest Process Next

- Cool idea if you know the remaining times
- exponential smoothing can be used to estimate a jobs' run time
- aT0 + (1-a)T1 where T0 and T1 are successive runs of the same job

Lottery Scheduling

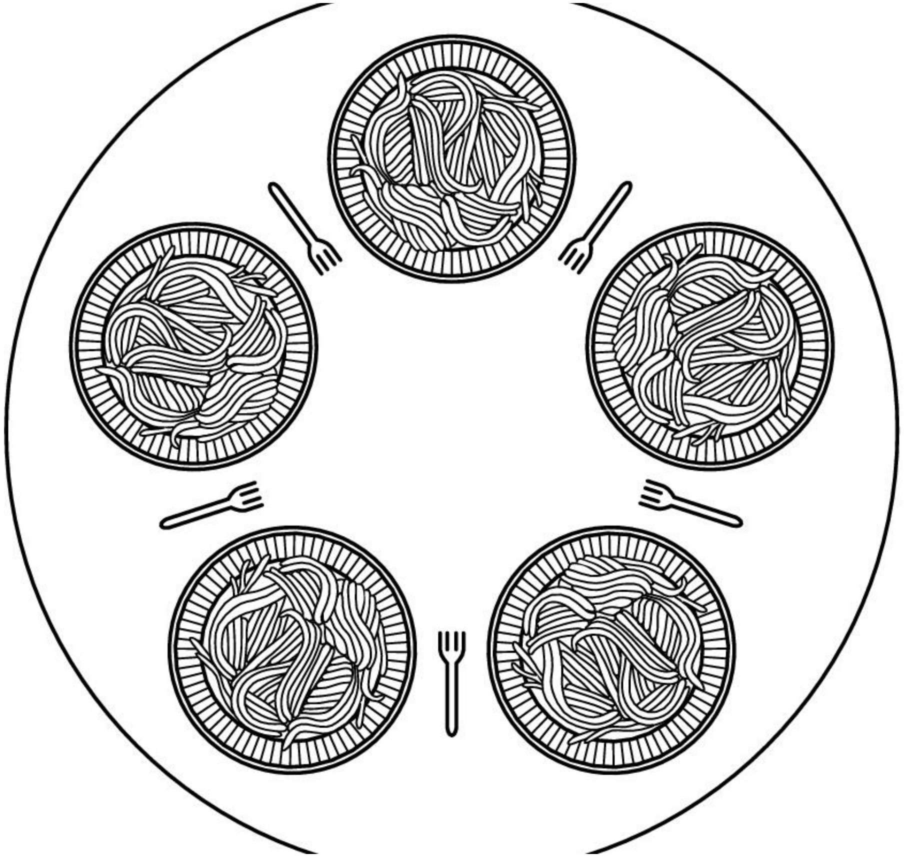
- Hold lottery for CPU time several times a second
- Can enforce priorities by allowing more tickets for “more important” processes

Real Time Scheduling

- Hard real time vs soft real time
  - Hard: robot control in a factory
  - Soft: CD player

- Events can be periodic or aperiodic
- Algorithms can be static (know run times in advance) or dynamic (run time decisions)

Dining Philosophers Problem (1)



. Lunch time in the Philosophy Department.

Dining Philosophers Problem (2)

```
#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 */
    }
}
```

A nonsolution to the dining philosophers problem.

Dining Philosophers Problem (3)

```
#define N           5              /* number of philosophers */
#define LEFT        (i+N−1)%N      /* number of i's left neighbor */
#define RIGHT       (i+1)%N        /* number of i's right neighbor */
#define THINKING    0              /* philosopher is thinking */
#define HUNGRY      1              /* philosopher is trying to get forks */
#define EATING      2              /* philosopher is eating */
typedef int semaphore;             /* semaphores are a special kind of int */
int state[N];                      /* array to keep track of everyone's state */
semaphore mutex = 1;               /* mutual exclusion for critical regions */
semaphore s[N];                    /* one semaphore per philosopher */

void philosopher(int i)            /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {                 /* repeat forever */
        think( );                  /* philosopher is thinking */
        take_forks(i);             /* acquire two forks or block */
        eat( );                    /* yum-yum, spaghetti */
        put_forks(i);              /* put both forks back on table */
    }
}
```

Dining Philosophers Problem (4)

```
void take_forks(int i)            /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);               /* enter critical region */
      state[i] = HUNGRY;          /* record fact that philosopher i is hungry */
      test(i);                    /* try to acquire 2 forks */
      up(&mutex);                 /* exit critical region */
      down(&s[i]);                /* block if forks were not acquired */
}
```

Dining Philosophers Problem (5)

```
void put_forks(i)                    /* i: philosopher number, from 0 to N-1 */
{
     down(&mutex);                   /* enter critical region */
     state[i] = THINKING;            /* philosopher has finished eating */
     test(LEFT);                     /* see if left neighbor can now eat */
     test(RIGHT);                    /* see if right neighbor can now eat */
     up(&mutex);                     /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N-1 */
{
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
           state[i] = EATING;
           up(&s[i]);
     }
}
```

A solution to the dining philosophers problem.

Takeaways