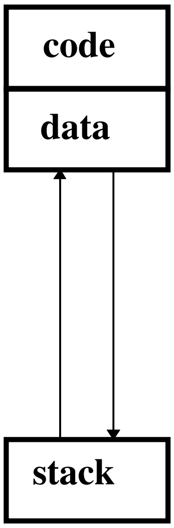
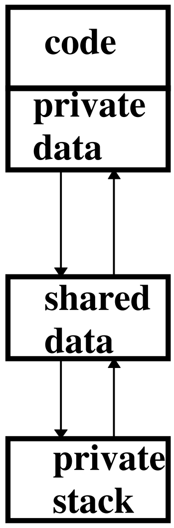
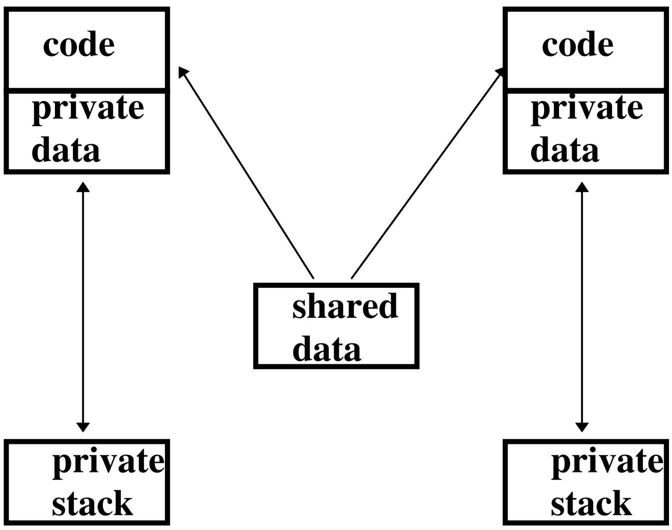
SHARED-MEMORY MODELS



UNIX serial process model



Sequent DYNIX process model



Two microtasks created by m_fork in DYNIX share some data

Parallel Programming Under DYNIX

SEQUENT C

- **The DYNIX operating system provides a set of routines to facilitate parallel processing.**

- **The commercial parallel programming languages the Sequent hardware uses are simple extensions of sequential languages that allow programmers to declare shared variables that interact via mutual exclusion and barrier synchronisation.**

Shared Data

- **Parallel processes on the Sequent coordinate their activities by accessing shared data structures.**

- **The keyword, shared, placed before a global variable declaration, indicates that all processes are to share a single instance of that variable, and changes made by one process can be detected by the other processes.**

  shared int a[10]

  On the other hand, if the array is declared

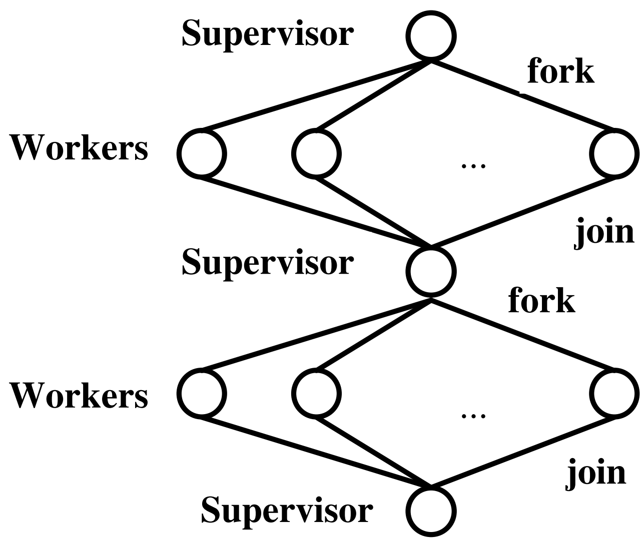
  int a[10]

  then every active process has its own copy of the array; if one process modifies a value in its copy of a, no other process's value will change.

- **The existence of a shared memory means that all process interactions can be performed through variables accessible to all the processes.**

Parallel Processing Functions

- **A program begins execution as a single process. This process is responsible for executing those parts of the program that are inherently sequential.**
- **When control reaches a part of the computation that may be performed in parallel, the original process forks a number of other processes, each process performing its share of the work.**
- **The total number of processes accessing shared data cannot exceed the number of physical processors less one hence, each process may execute on its own CPU.**
- **When control reaches an inherently sequential portion of the computation, only the original process executes the code; the remaining processes wait until control reaches another portion of the computation that can be executed concurrently. The program cycles through these two modes until termination.**



- **The transition from parallel to sequential execution is always delimited by a *barrier synchronisation*. No process may proceed beyond a barrier synchronisation point until every process has reached the barrier.**

Sequent's multitasking library

Two new types are needed for locking and setting up barriers. Variables of type **slock_t** are used to set and clear **access locks**, and variables of type **sbarrier_t** are used to set and clear **barriers**. The appropriate operations for objects of these types are listed below.

Operations on variables of type slock_t:

s_init_lock():	initialise a lock variable
s_lock():	set the lock
s_unlock():	clear the lock

Operations on variables of type sbarrier_t:

s_init_barrier():	initialise the barrier variable
s_wait_barrier():	wait for all multitasks to reach barrier

The usual **fork()** and **cpus_online()** routines are used to create a multitask and query the system for the number of processors, respectively.

Sequent's microtasking library

- *m_set_procs(p):* **By calling m_set_procs with argument p, the parent process initialises to value p a shared variable that controls the number of processes created by a subsequent call to m_fork. The value of p cannot exceed the number of physical processors in the system minus one. The function also initialises barriers and locks.**

- *m_fork(name [,arg,...]):* **The parent process creates a number of child processes, then the parent process and the child processes begin executing function name with the arguments (if any) also specified by the call to m_fork. After all the processes (the parent and all the children) have completed execution of function name, the parent process resumes execution with the code after m_fork, while the child processes busy wait until the next call to m fork. Therefore, the first call to m fork is more expensive than subsequent calls, because only the first call entails process creation.**

- *m_get_myid:* **A process calls function m_get_myid to get its unique process number. If the total number of active processes is p, then the process number of the parent is 0, while the process numbers of the child processes range from 1 to p-l.**

- *m_get_numprocs:* **Function m_get_numprocs returns the number of processes executing in parallel. Given the total number of processes and its own process number, a process can determine which portion of a computation is its responsibility.**

- *m_lock, m_unlock:* **Functions m_lock and m_unlock ensure mutually exclusive execution of the code that the two calls surround. Once a process has entered a block of code delimited by m_lock and m_unlock, no other process may enter until the first process has left.**

- *m_kill_procs:* **Function m_kill_procs kills the child processes created by the first call to m_fork.**

Operations on variables of type sbarrier_t:

- **m_sync(): wait for all microtasks to arrive**
- **m_single(): wait for supervisor thread to call m_multi()**
- **m_multi(): resume all workers in parallel**

ILLUSTRATING EXAMPLE

The program creates one supervisor, and nprocs-1 workers. The code for each worker is given as function work(). The example illustrates both methods of synchronising access to a shared variable.

```
/* Global declarations */
shared int x;
shared struct y_struct {
   int state;
   slock_t lp;          /* primitive lock */
   } y;

/* Supervisor microtask */
main() {
   s_init_lock (&y.lp);  /*initialize primitive lock */
   m_set_procs (nprocs); /*initialize nprocs-1 workers*/
   m_fork (work);        /*create and run nprocs-1 microtasks*/
   m_kill_procs ();      /*done, so kill worker microtasks*/
   }

/* Worker microtask */
work() {
   /* illustrates microtask locking */
   m_lock();   /* begin mutual exclusive access to x */
   x++;        /* increment shared variable x */
   printf ("worker");    /* just a test message */
   m_unlock();           /* end of critical section */

   /* illustrates a barrier */
   m_single();           /* all workers wait here until m_multi */
   printf ("Supervisor"); /* only supervisor executes this */
   m_multi();             /* resume all workers ... */

   /* illustrates multitask locking */
   s_lock (&y.lp);         /* lock structure y */
   y.state = 0;            /* change the structure */
   printf ("structure");   /* another test message */
   s_unlock (&y.lp);       /* unlock structure y */
   }
```

Sequent C program to compute π using numerical integration

```
#include < stdio.h>
#include < parallel/microtask.h>
#include < parallel/parallel.h>
shared double pi;                  /* Approximation to pi */

main (argc, argv)
        int argc; char *argv[];
{
        void computepi();
        int intervals;             /* Number of intervals */
        int numprocs;              /* Number of processes to be forked */
        numprocs = atoi (argv[1]);
        intervals = atoi (argv[2]);
        m_set_procs (numprocs);
        pi = 0.0;
        m_fork (computepi, numprocs, intervals);
        printf ("Estimation of pi is %14.12f.\n", pi);
        m_kill_procs ();
        return();
}

void computepi (numprocs, intervals)
        int numprocs, intervals;
{
        int id;                    /* Index of this process */
        int i;
        double localsum, width, x;
        id = m_get_myid ();
        localsum = 0.0;
        width = 1.0/intervals;
        for (i = id; i < intervals; i += numprocs){
                x = (i+0.5)*width;
                localsum += (4.0/(l.0+x*x));
        }
        localsum *= width;
        m_lock ();
        pi += localsum;
        m_unlock ();
}
```

ANALYSIS OF SHARED-MEMORY PROGRAMS

- A **lock** is a mechanism for enforcing a policy for sharing access to data.
- A **policy** is a set of rules that define allowable values for shared variables.
- Determinism of a policy

    1. **determinism**: access order is always the same no matter how many times the program is run.

    2. **nondeterminism**: access order is left up to chance, and so may be different each time the program is run.

    3. **indeterminacy**: the result of nondeterminism, where the value of a shared variable is left to chance. Each time the program is run, it is possible that a different value is computed for the shared variable.

- A **safe policy** is one that enforces deterministic results.
- A **safe lock** is one that enforces determinism.
    - **Mutual exclusion** is a lock that enforces a safe policy because only one task is permitted to access the shared resource at a time.
- A **policy is unfair** if it permits one task to wait indefinitely while other tasks repeatedly access shared data.
- A **fair lock** guarantees access to all tasks that have requested access, before second or third accesses are granted to any pending tasks. That is, everyone gets access before anyone gets multiple accesses.
- **Deadlock** is a state where none of the tasks can continue on because they mutually block each other.
- A **policy is live** if it prevents deadlock.
- A **lock is live** if it is impossible for the lock to indefinitely postpone access by waiting tasks.
    - A mutual exclusion lock may on its own be live, but when used in combination with other mutual exclusion locks, it may cause a **system deadlock**.